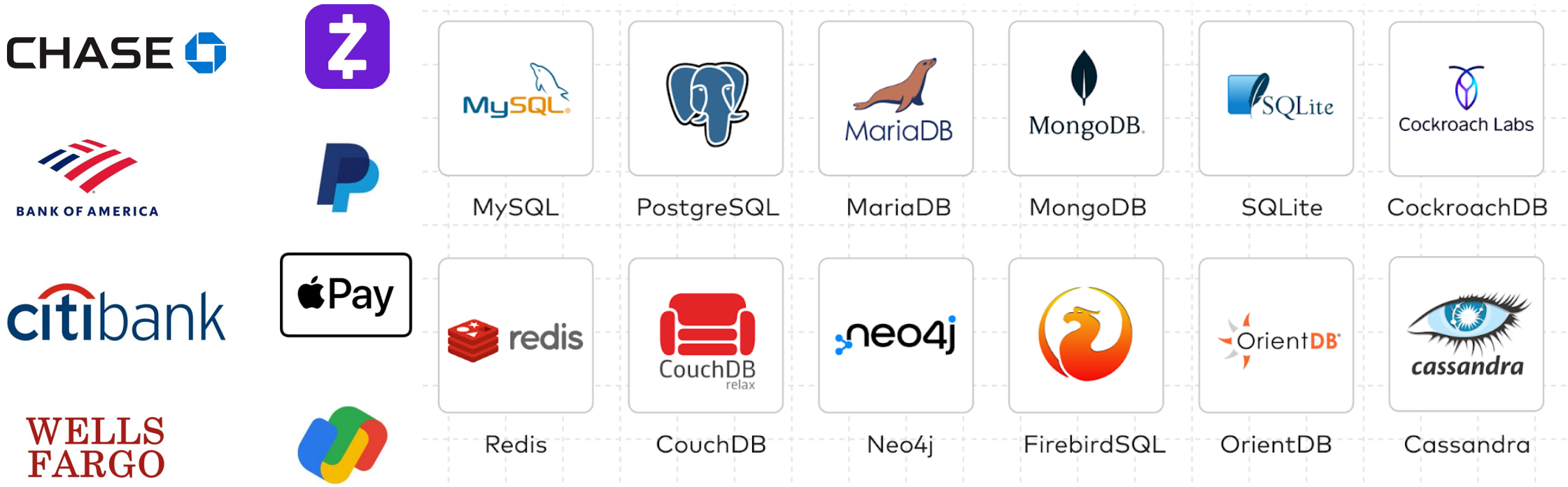
Recovery can be Simple: Asynchronous Logging for Distributed Transactions

Sky Summer Retreat – 2024

Soujanya Ponnapalli
Mentor: Jonathan Goldstein

Applications and Distributed Transactions

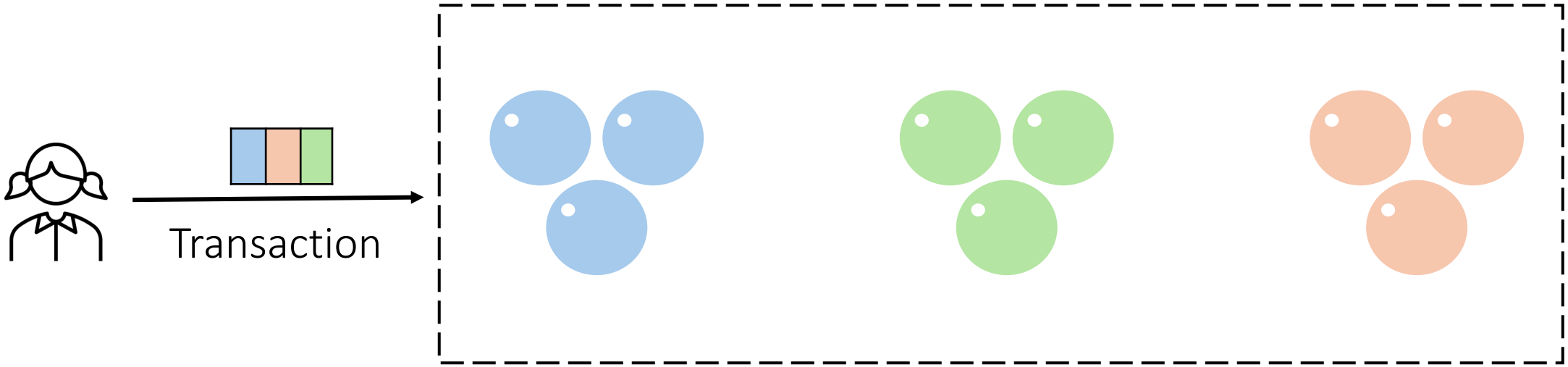


```
~$ touch sky-summer-retreat.txt
~$ mv sky-summer-retreat.txt sky-summer-retreat-24.txt
~$
```

Distributed Transactions have low throughput and poor scalability!

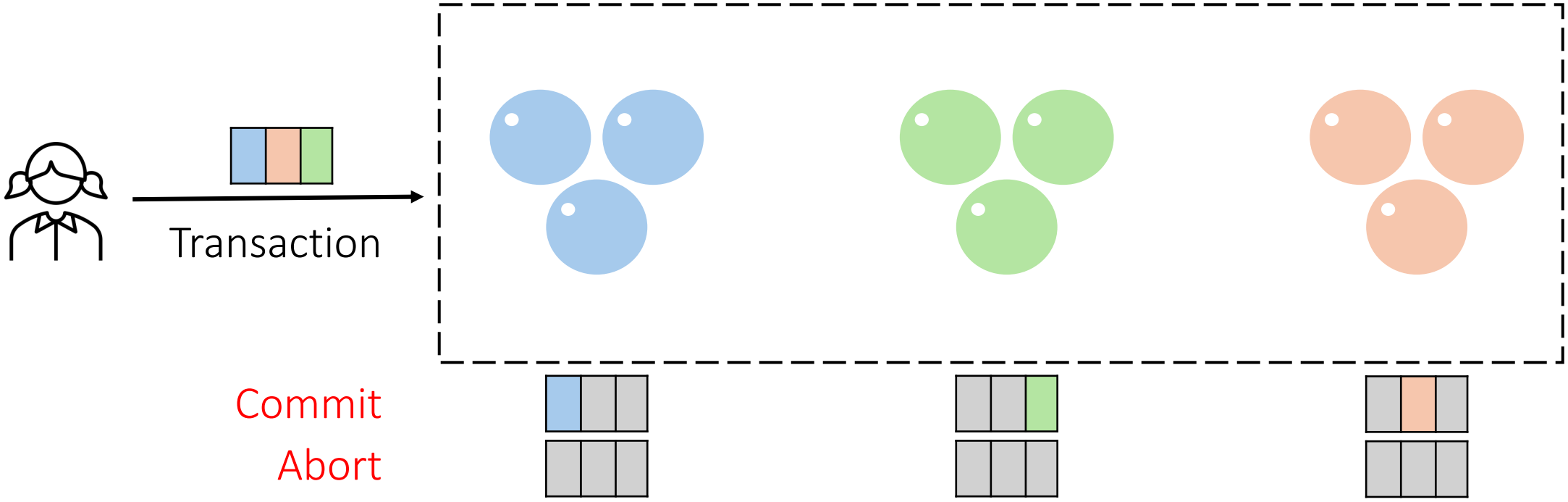
Back to Fundamentals: Distributed Transactions

In the context of databases



Back to Fundamentals: Distributed Transactions

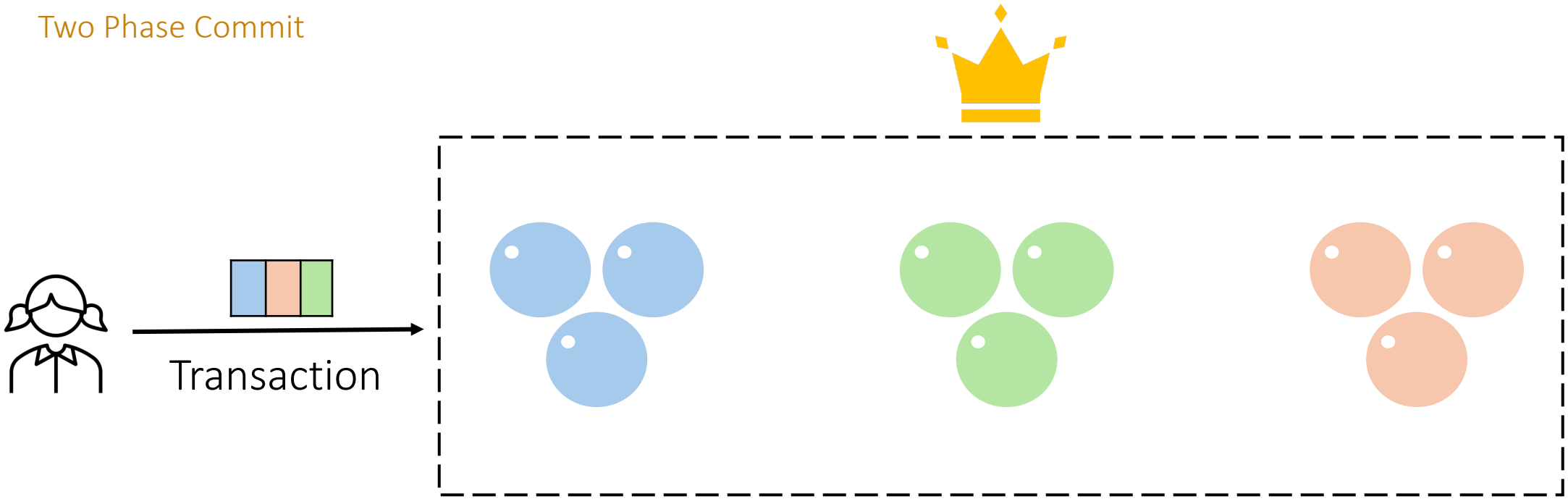
In the context of databases



For atomicity, databases rely on distributed commit protocols

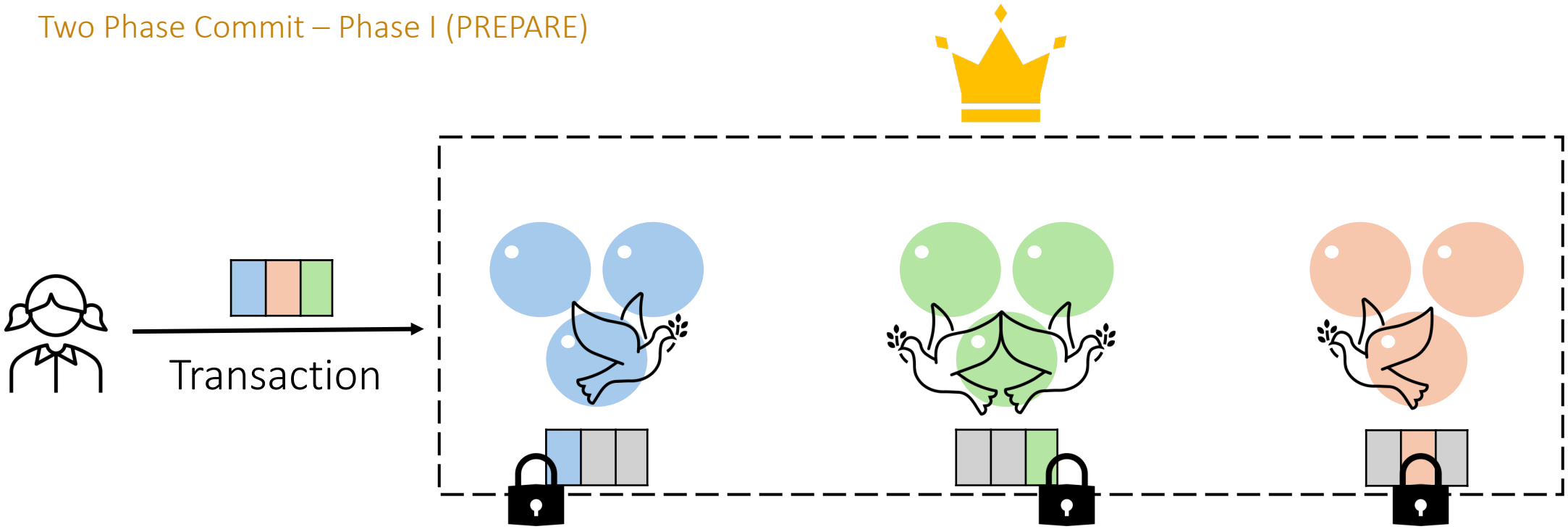
Back to Fundamentals: Distributed Transactions

Two Phase Commit



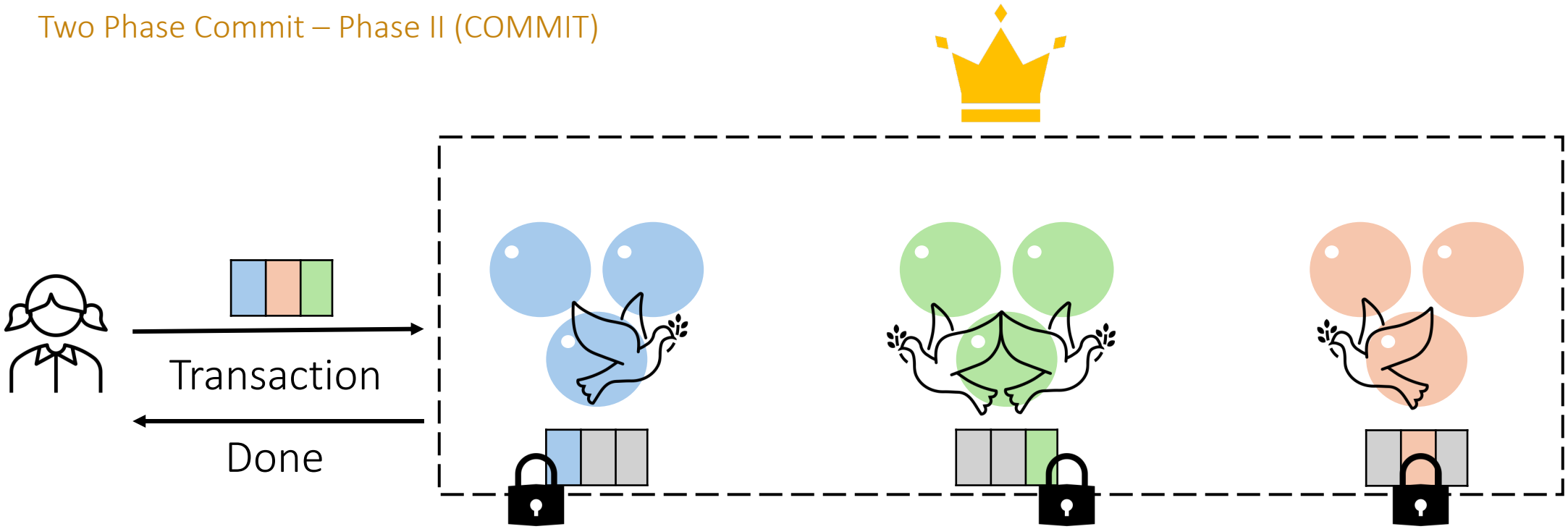
Back to Fundamentals: Distributed Transactions

Two Phase Commit – Phase I (PREPARE)



Back to Fundamentals: Distributed Transactions

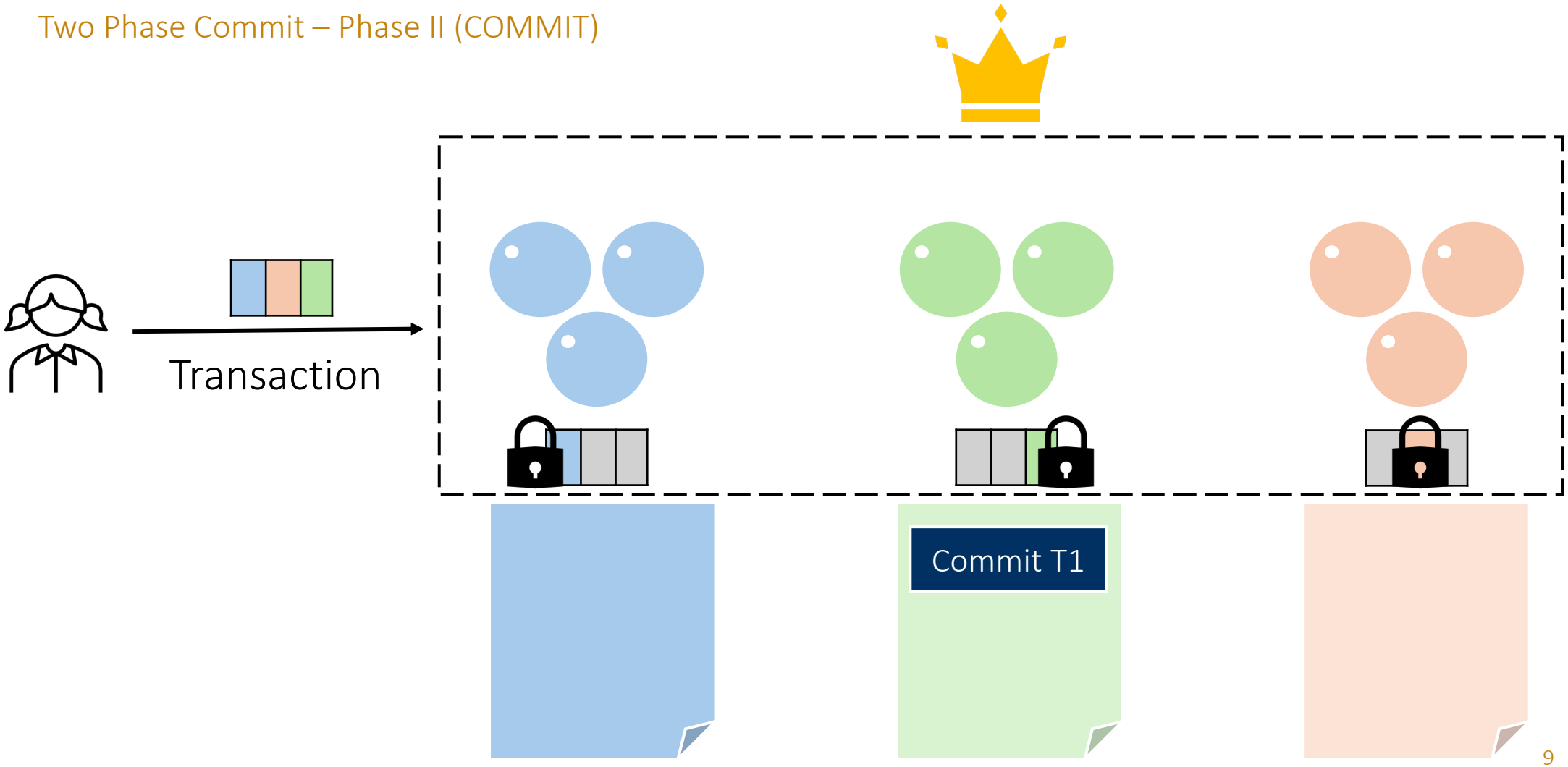
Two Phase Commit – Phase II (COMMIT)



To provide fault-tolerant ACID transactions, databases use logs!

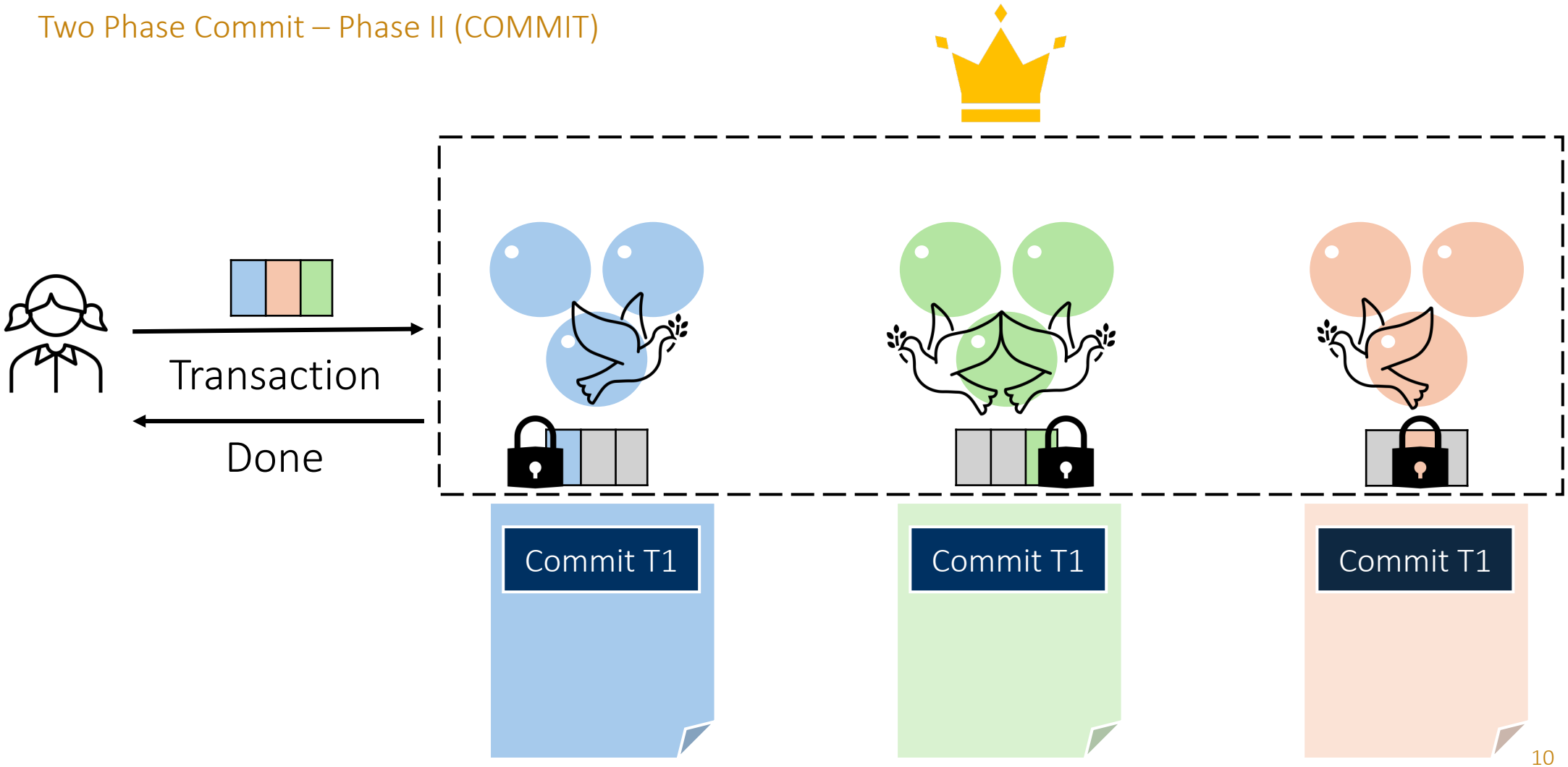
Back to Fundamentals: Distributed Transactions

Two Phase Commit – Phase II (COMMIT)



Back to Fundamentals: Distributed Transactions

Two Phase Commit – Phase II (COMMIT)



Updating Recovery Logs in the Critical Path!

Transaction Commit Path:

1. Coordinator **writes to its Recovery Log** before sending commit msgs
2. Servers **write to their Recovery Logs** before acknowledging commit msgs
3. Primary notifies the client

Transaction throughput is limited by the I/O throughput at the servers!

Highly-contented transactions stall for I/O to complete and scale poorly!

Logging Limits Performance – Why Bother? ☹

Two-orders of performance difference between storage and network

Networks got faster!

- Accelerated networking with low-latency NICs
- General-purpose datacenter networks like eRPC
- ~100x higher throughput than gRPC

For two servers in the same datacenter at Azure

- eRPC could handle about 2.6 Mops/s
- Whereas network-replicated disks could support 30 Kops/s

Mitigating I/O Bottlenecks: Early Lock Release

IMPLEMENTATION TECHNIQUES FOR MAIN MEMORY DATABASE SYSTEMS

David J. DeWitt[1], Randy H. Katz[2], Frank Olken[3],
Leonard D. Shapiro[4], Michael R. Stonebraker[2], David Wood[2]

[1] Computer Sciences Department, University of Wisconsin
[2] EECS Department, University of California at Berkeley
[3] CSAM Department, Lawrence Berkeley Laboratory
[4] Department of Computer Science, North Dakota State University

This research was partially supported by the National Science Foundation under grants MCS82-01360, MCS82-01870, by the Department of Energy under contracts #DE-AC02-81ER10920, #DE-AC03-76SF00098, #W-7405-ENG-48, and by the Air Force Office of Scientific Research under Grant 83-0021.

Partial Strictness in Two-Phase Locking

Eljas Soisalon-Soininen and Tatu Ylönen

Department of Computer Science, Helsinki University of Technology
Otakaari 1, FIN–02150 Espoo, Finland
e-mail: ess@cs.hut.fi, ylo@cs.hut.fi
telefax: +358-0-451 3293, tel: +358-0-4511

Abstract. Two-phase locking is a standard method for managing concurrent transactions in database systems. In order to guarantee good recovery properties, two-phase locking should be strict, meaning that locks can be released only after the transaction's commit or abort. In this paper we show that even exclusive locks can be released immediately after the *commit request* has arrived, without sacrificing any important recovery properties. This optimization is especially useful if the commit operation takes much time compared with the other actions, as for main-memory databases, or if the commits are performed in batches.

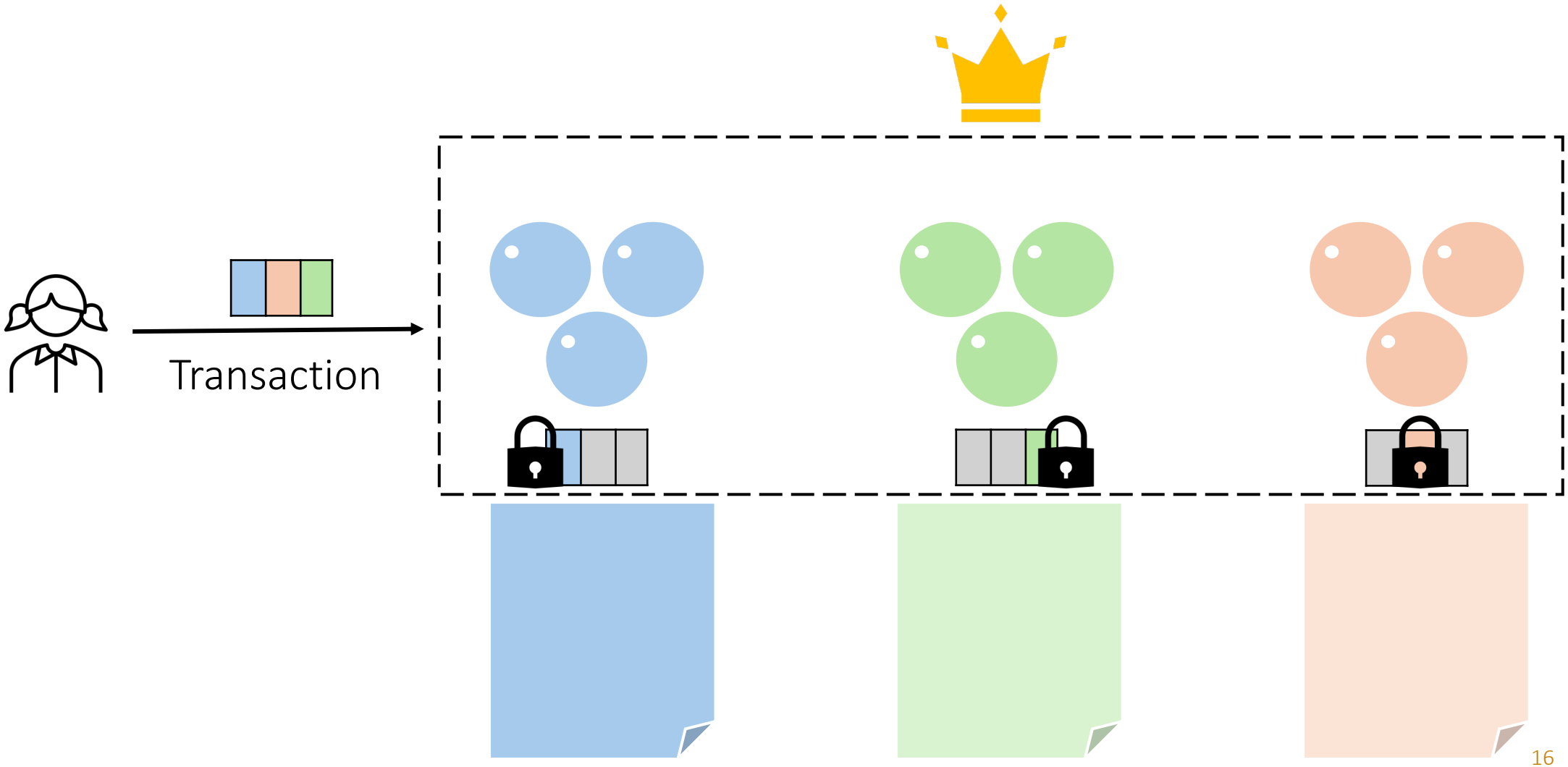
Mitigating I/O Bottlenecks: Early Lock Release

Limited wide-adoption of ELR and asynchronous logging

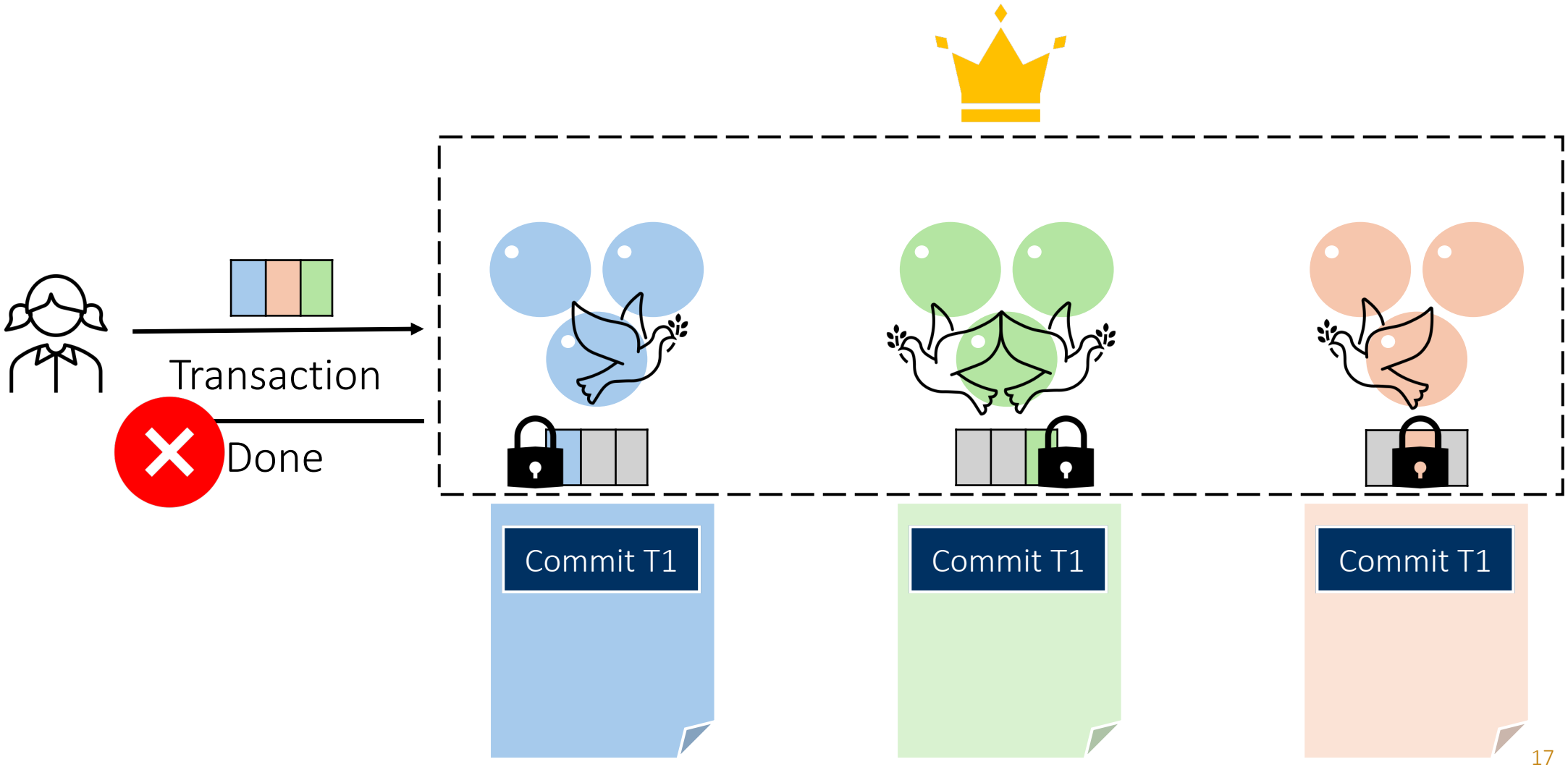
- In-consistent in-memory state
- Crash recovery and consistency

An active tradeoff between achieving high performance and keeping recovery simple!

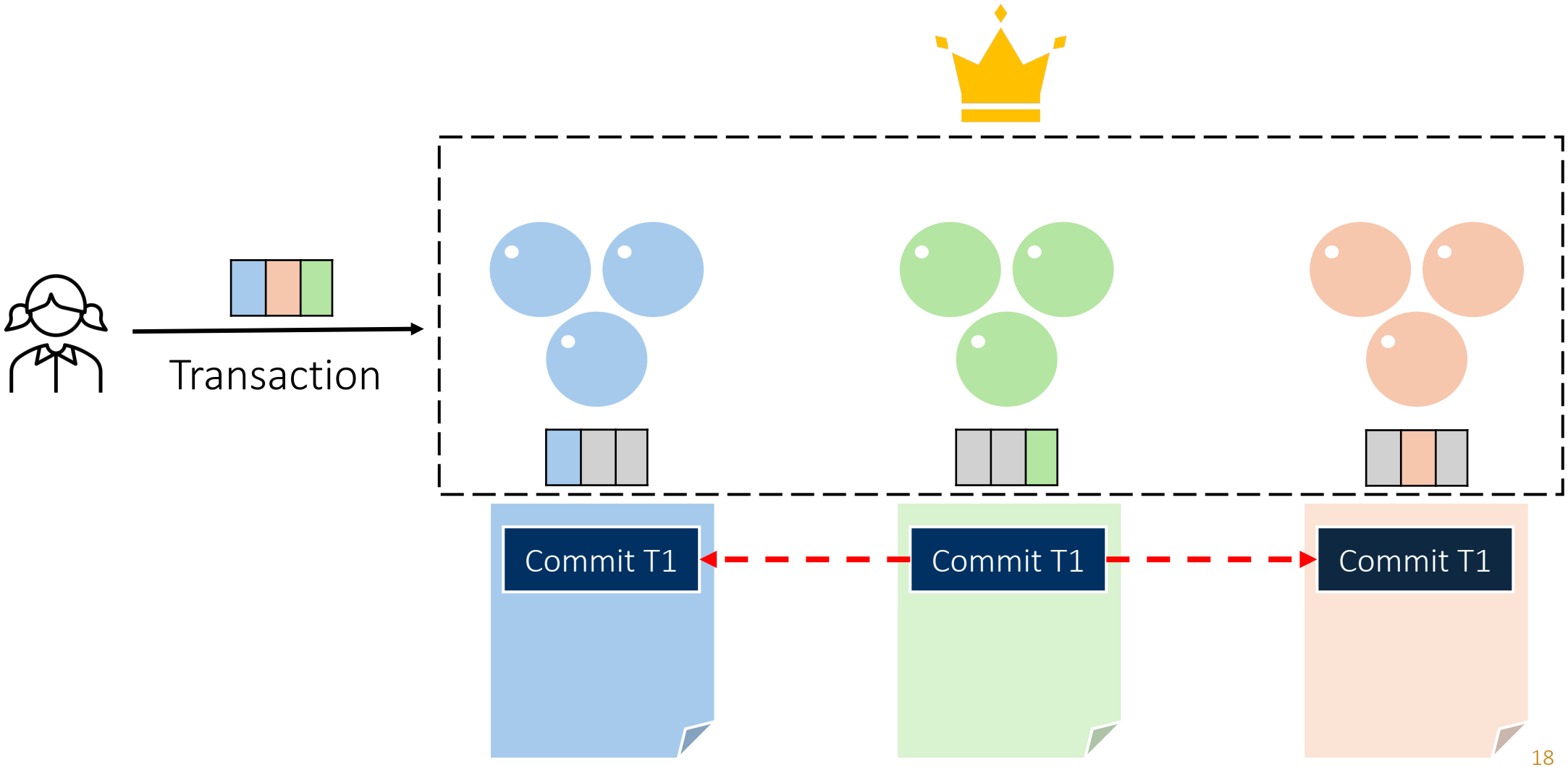
Simple Idea: Asynchronous Logging



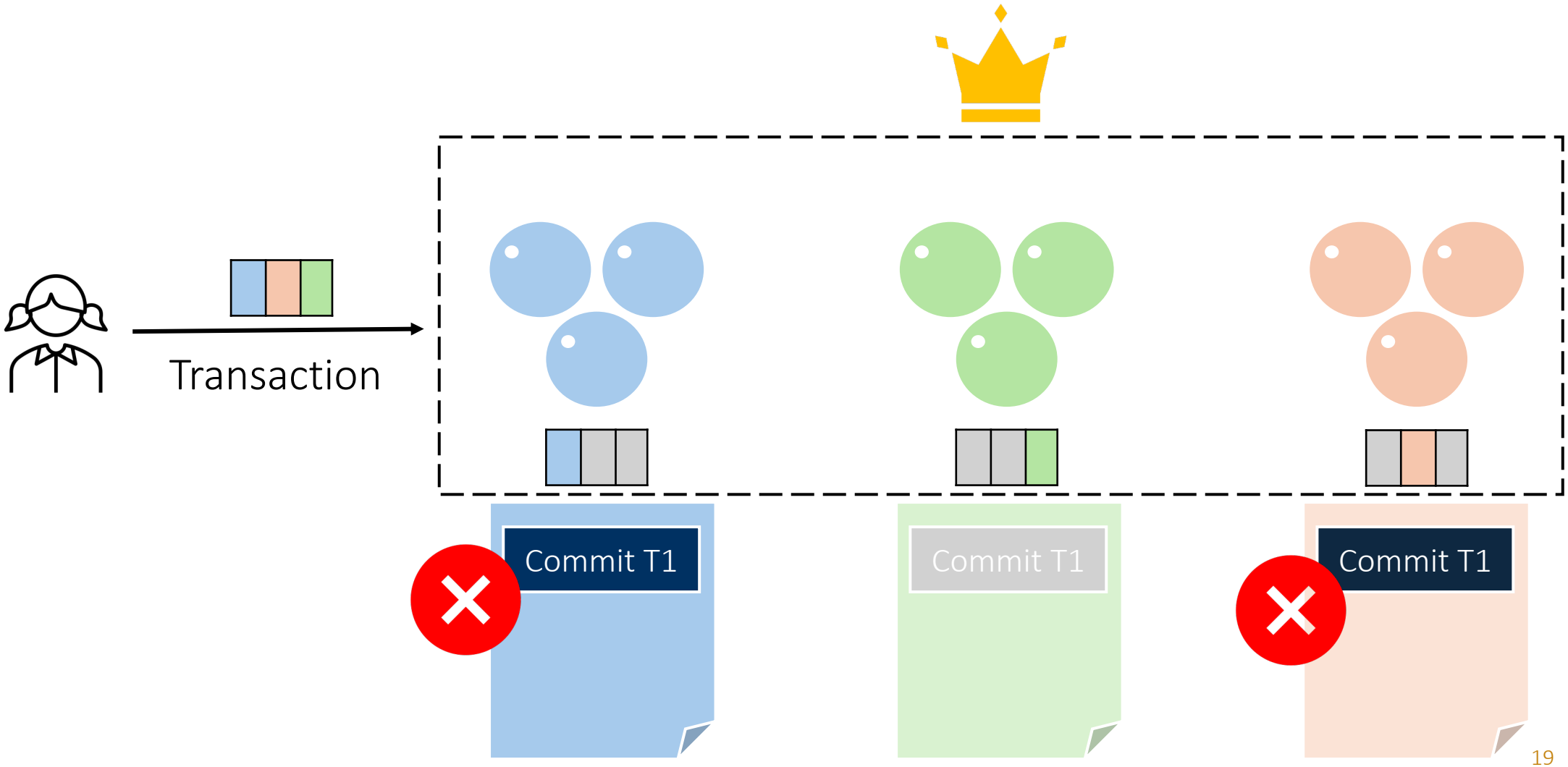
Simple Idea: Asynchronous Logging



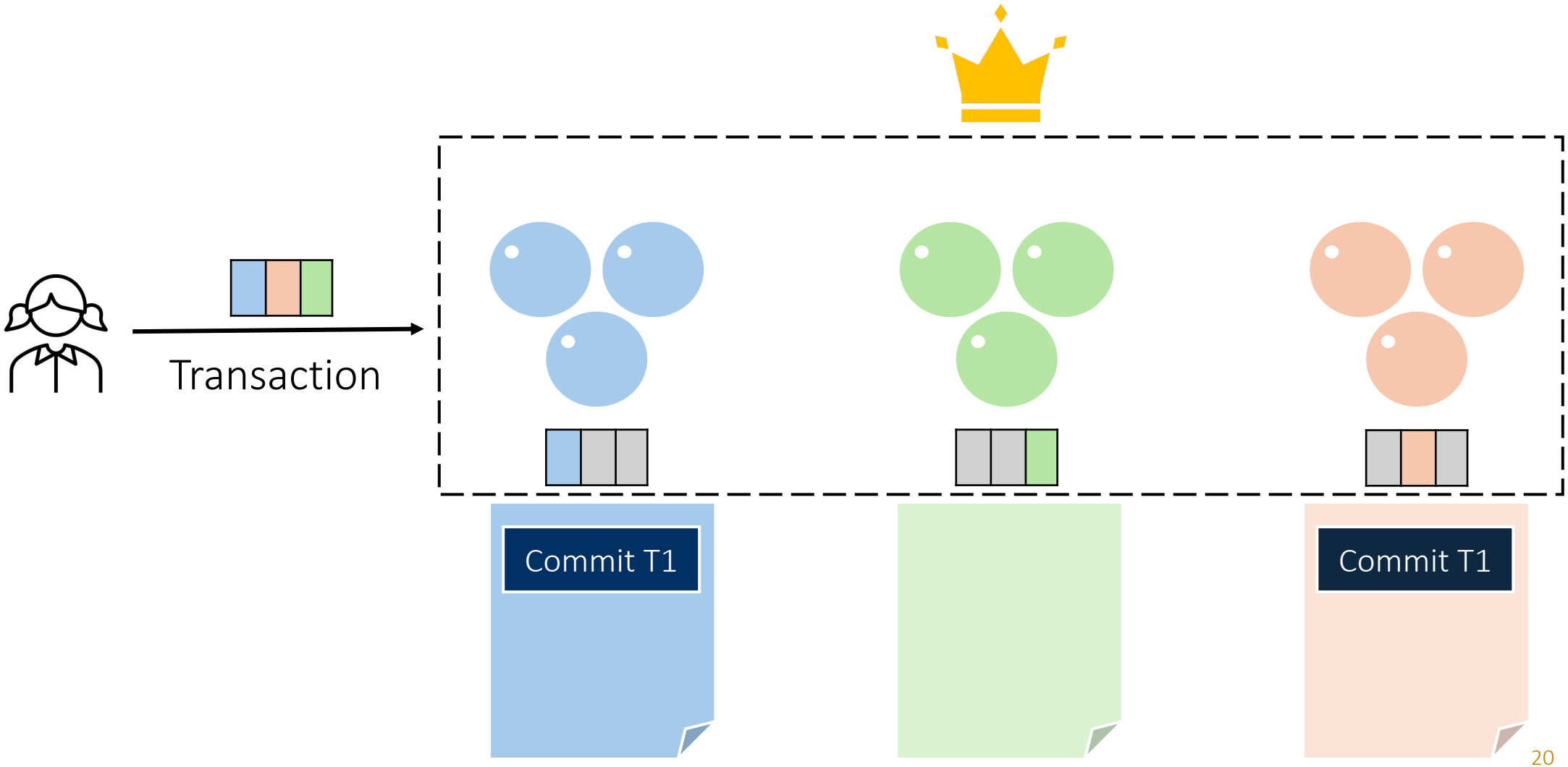
Simple Recovery: Fundamental Challenge



Simple Recovery: Fundamental Challenge



Simple Recovery: Fundamental Challenge



Recovery Becomes Complicated!

Needs tracking distributed dependencies across Recovery Logs

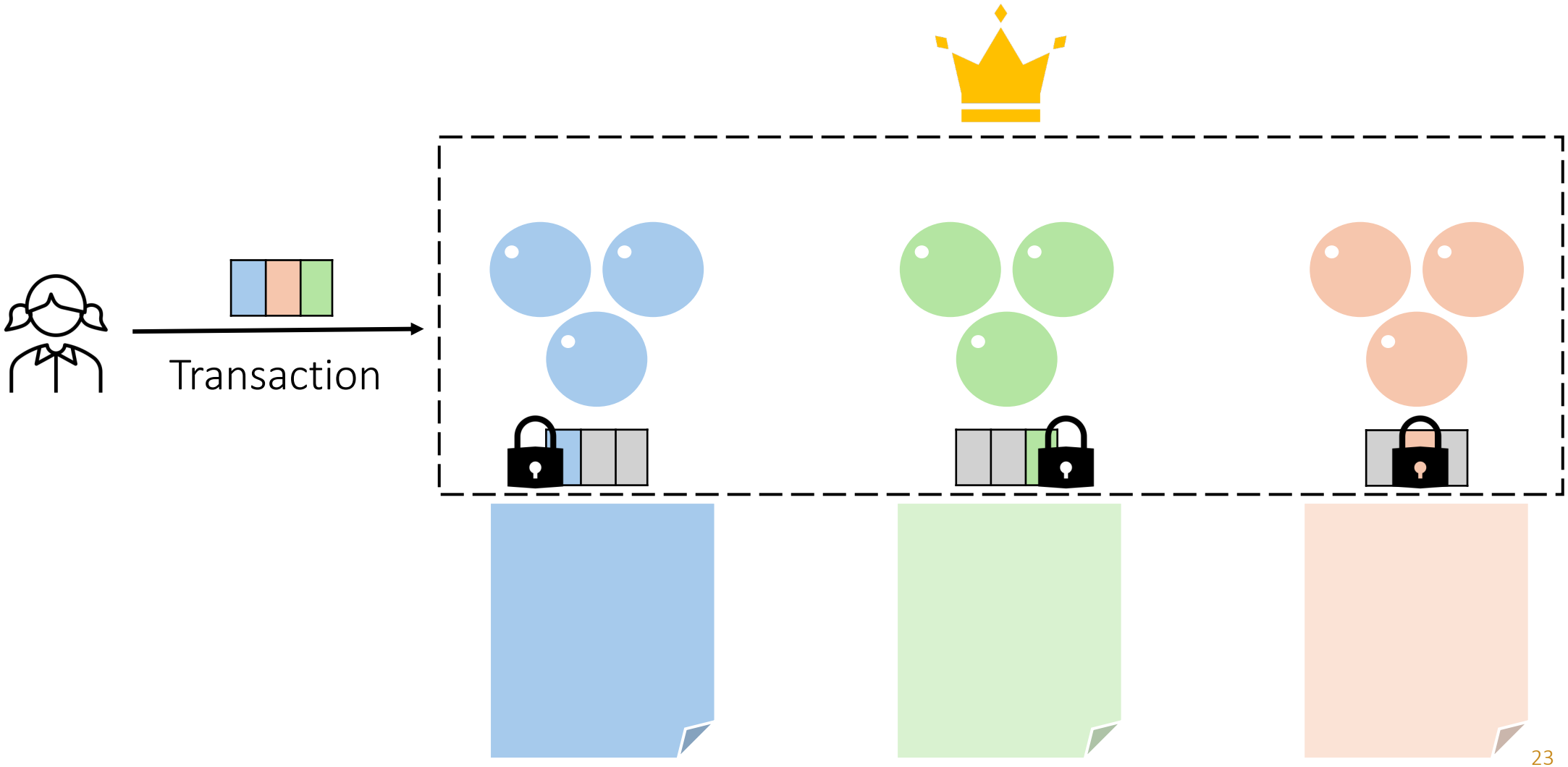
Too many possible states to potentially recover from

Cascades: Recovery Can Be Simple!

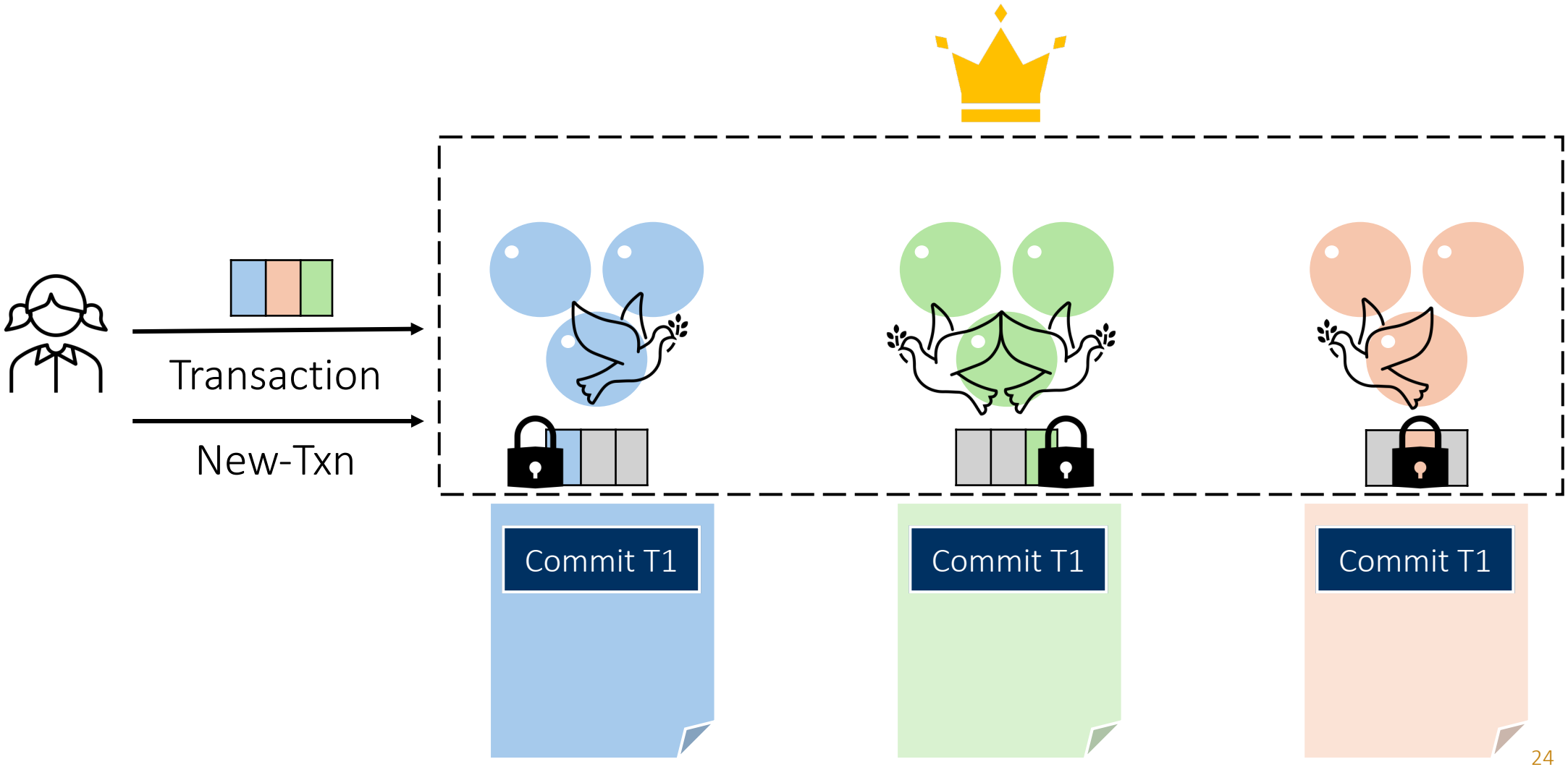
- Persistence vs Durability
- Persistence ☑
  - Commit record is flushed to disk
- Durability ☑
  - Commit record and all its dependencies are persisted

Commit record is durable when it and its dependencies are persisted

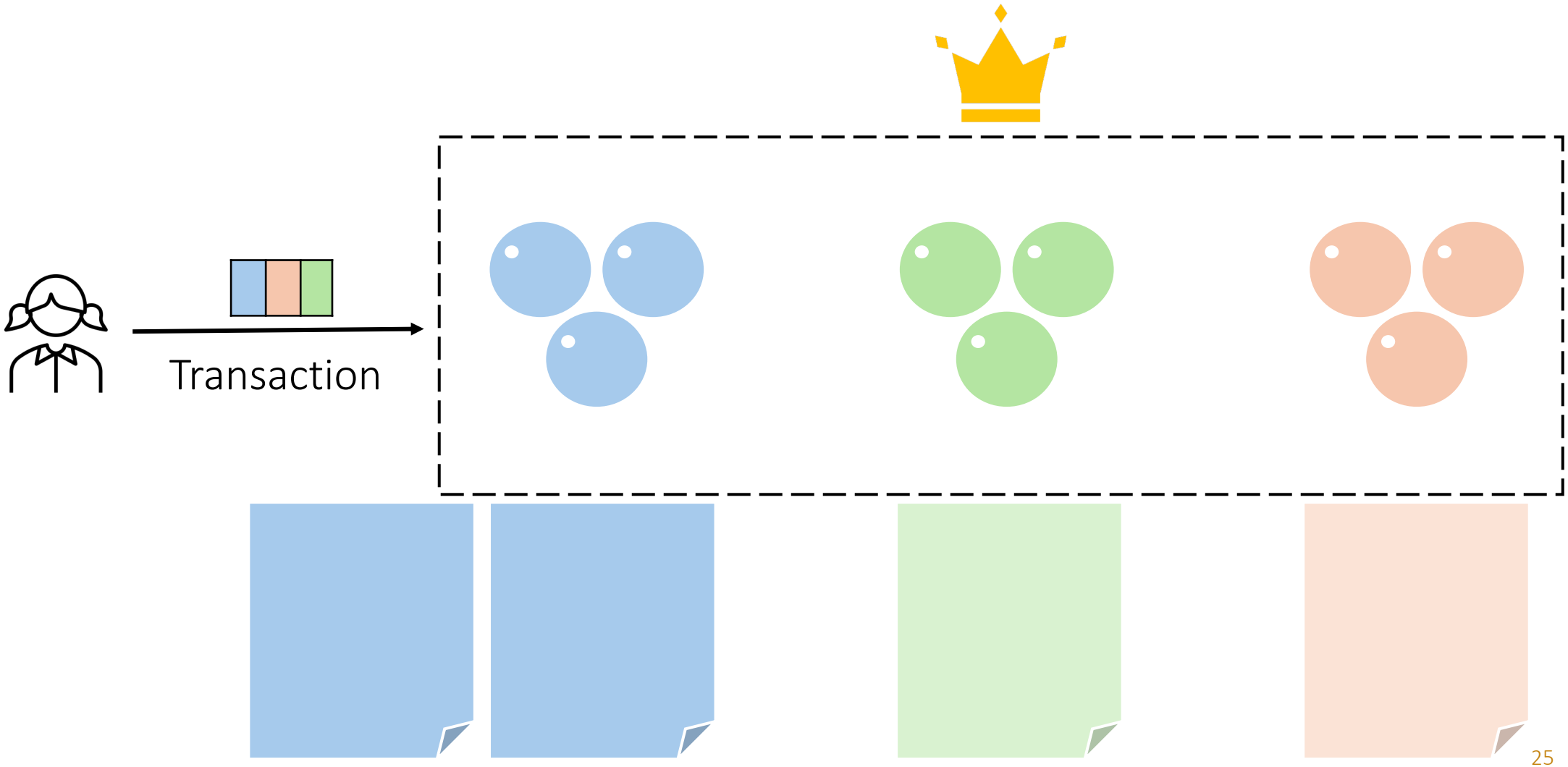
Insight: Speculate on Durability of Commit Records



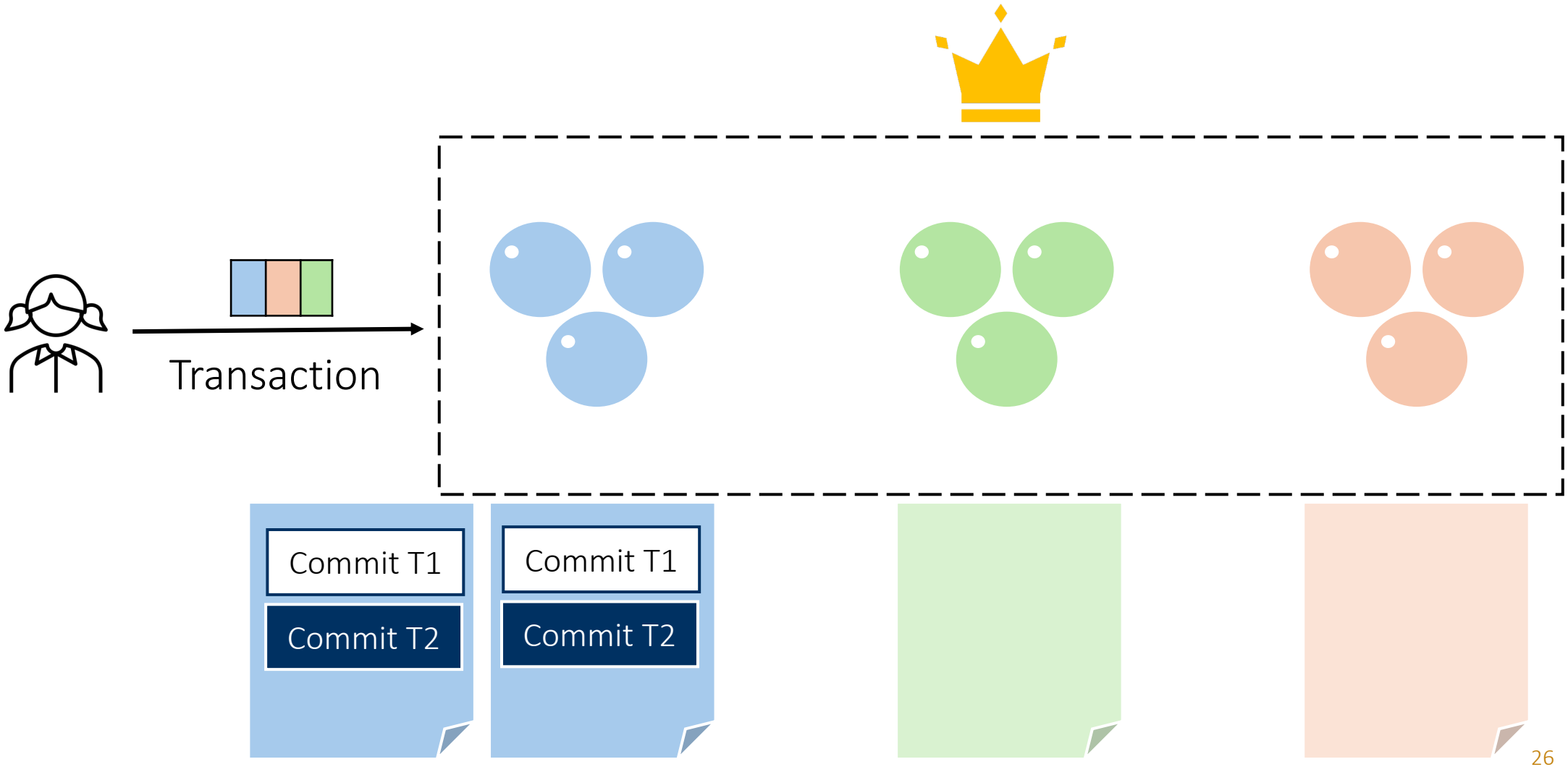
Insight: Speculate on Durability of Commit Records



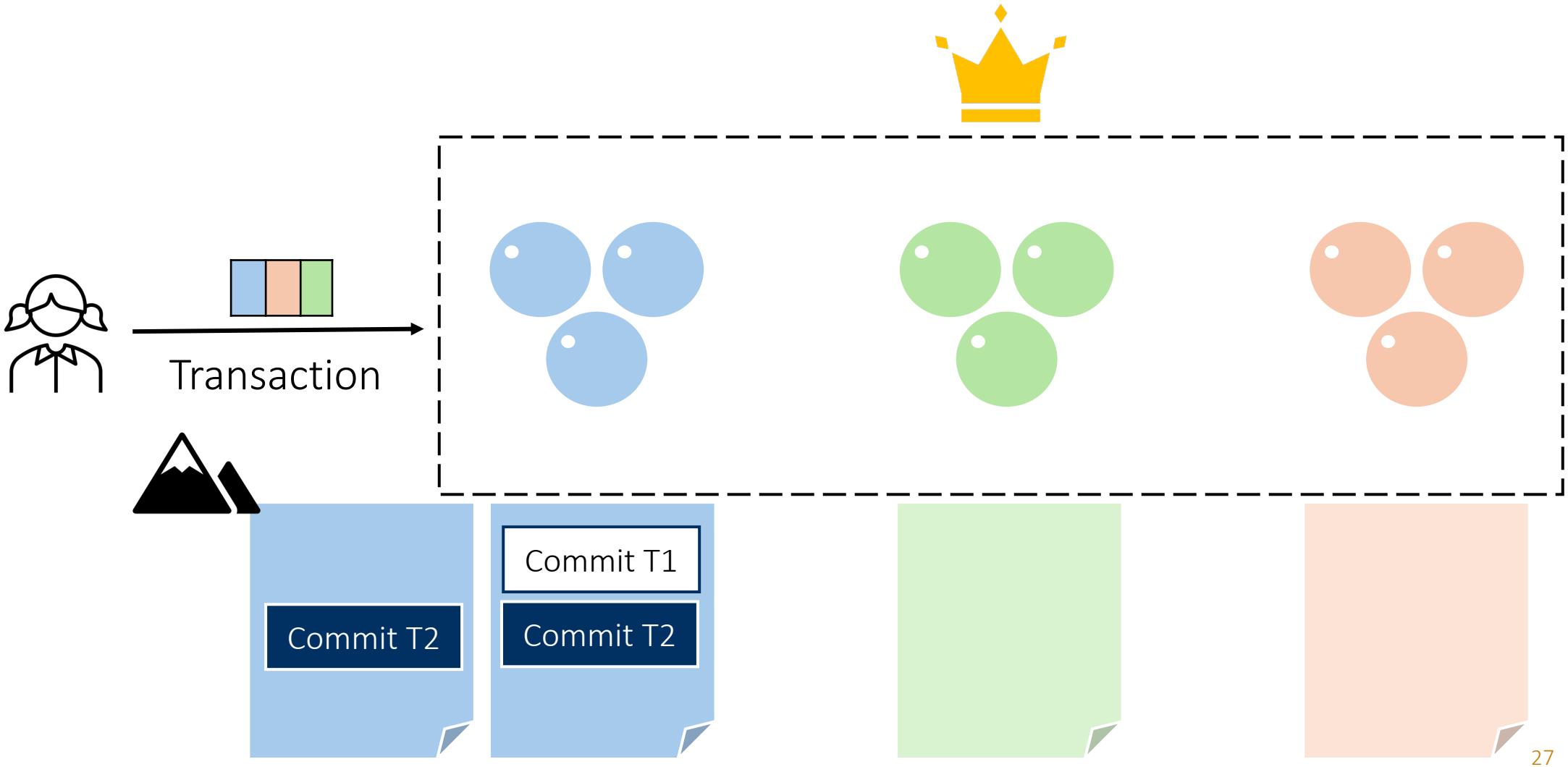
Insight: Replicate Durable Records Asynchronously



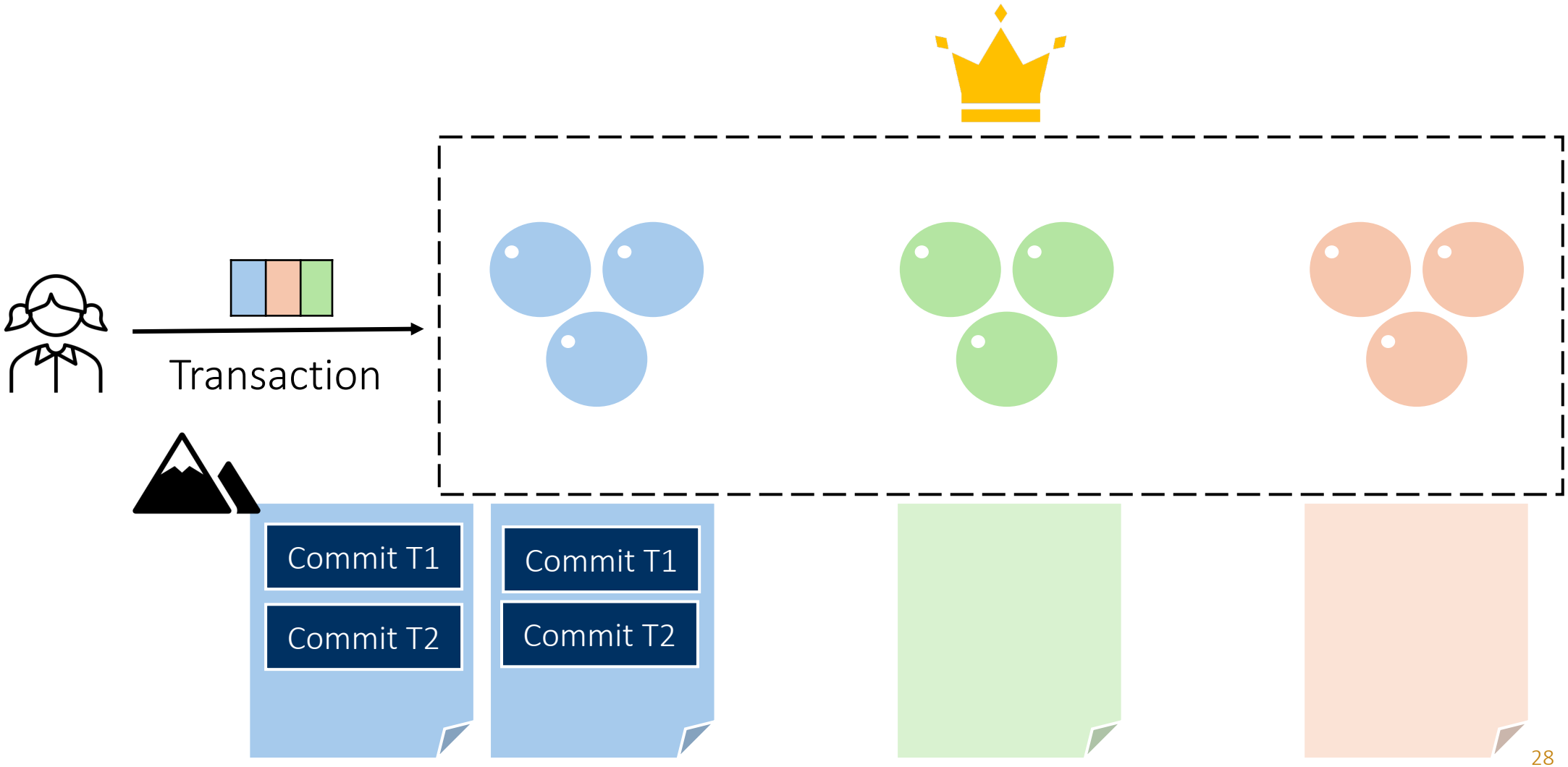
Insight: Replicate Durable Records Asynchronously



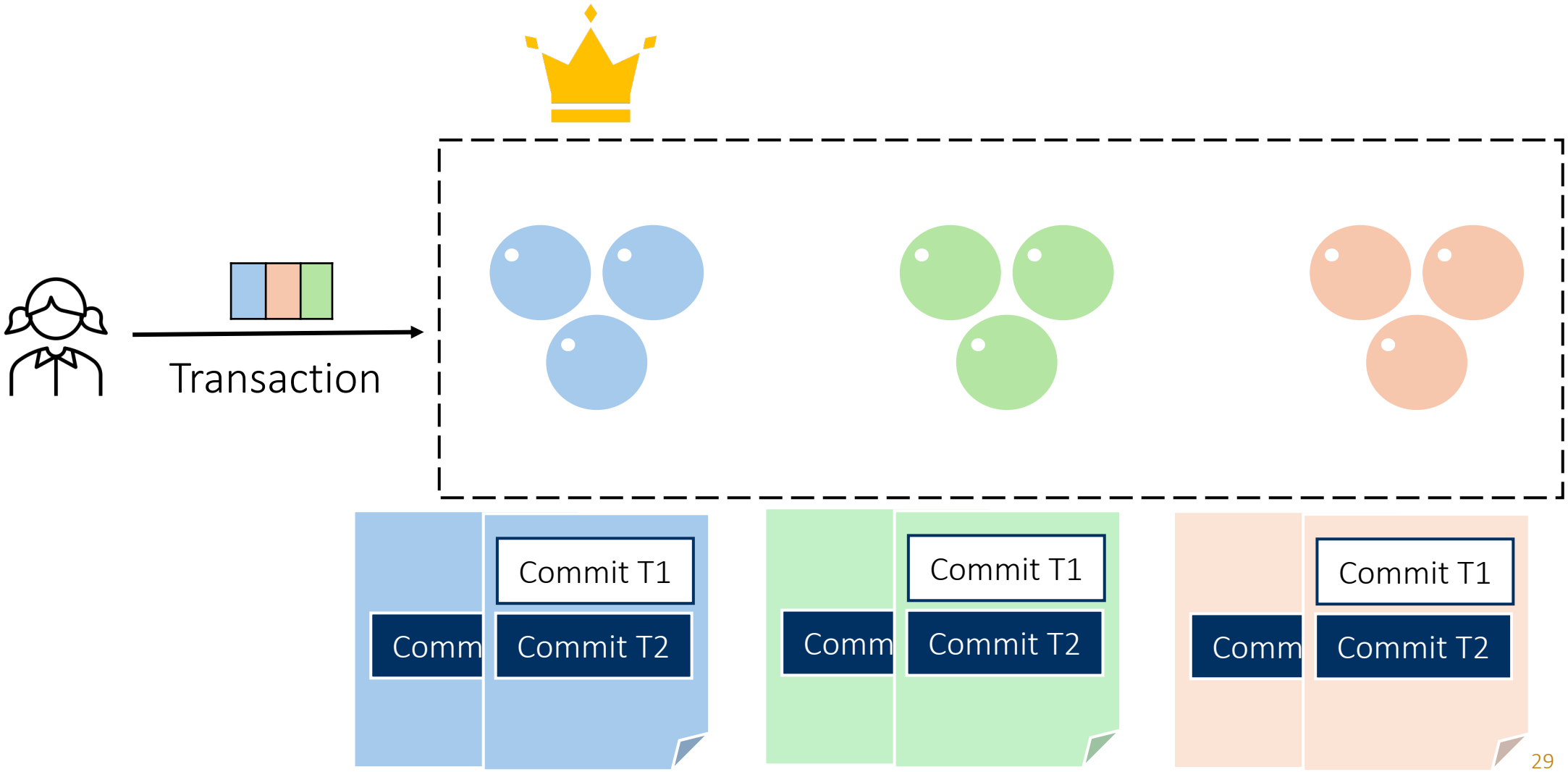
Insight: Replicate Durable Records Asynchronously



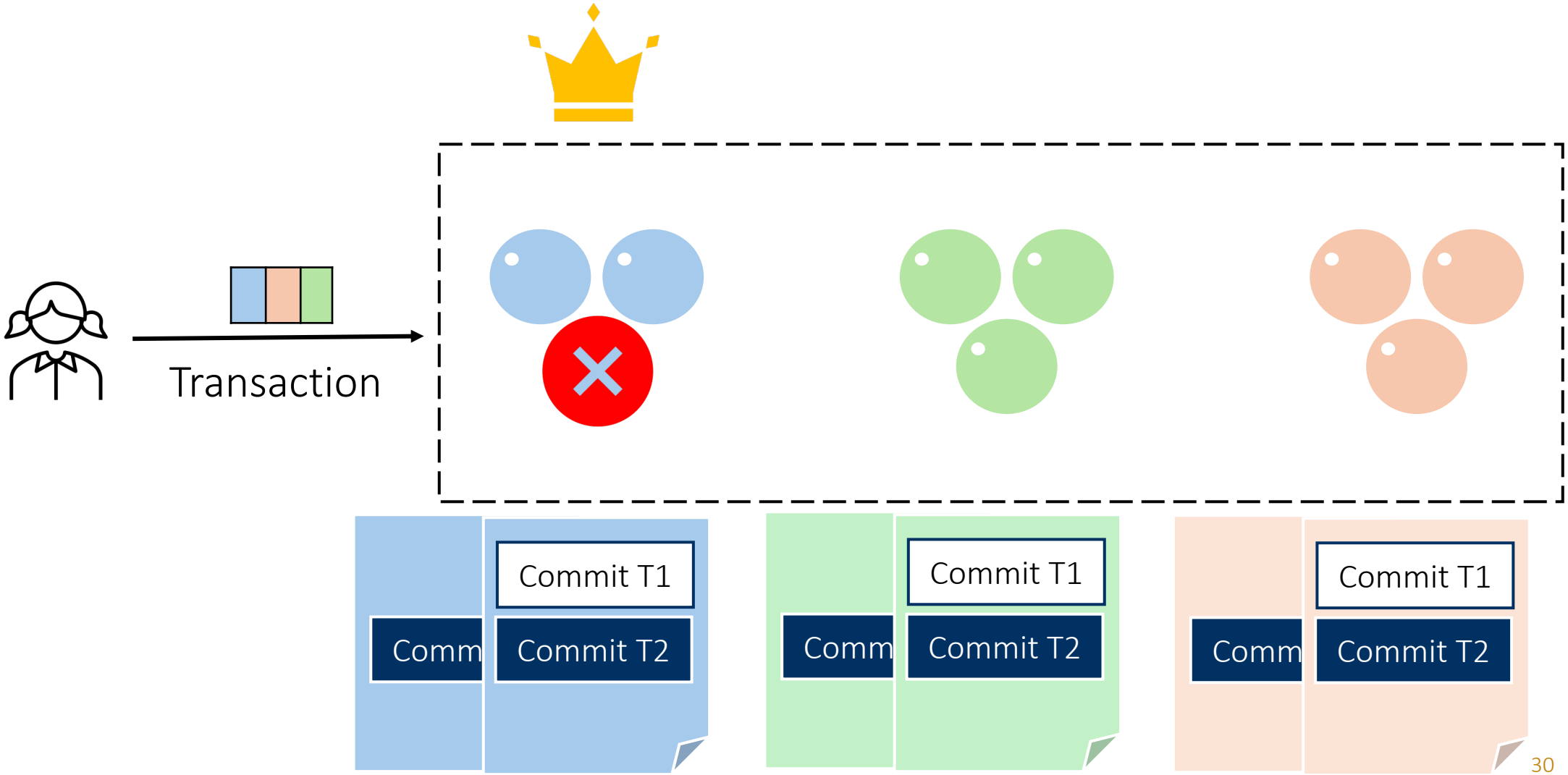
Insight: Replicate Durable Records Asynchronously



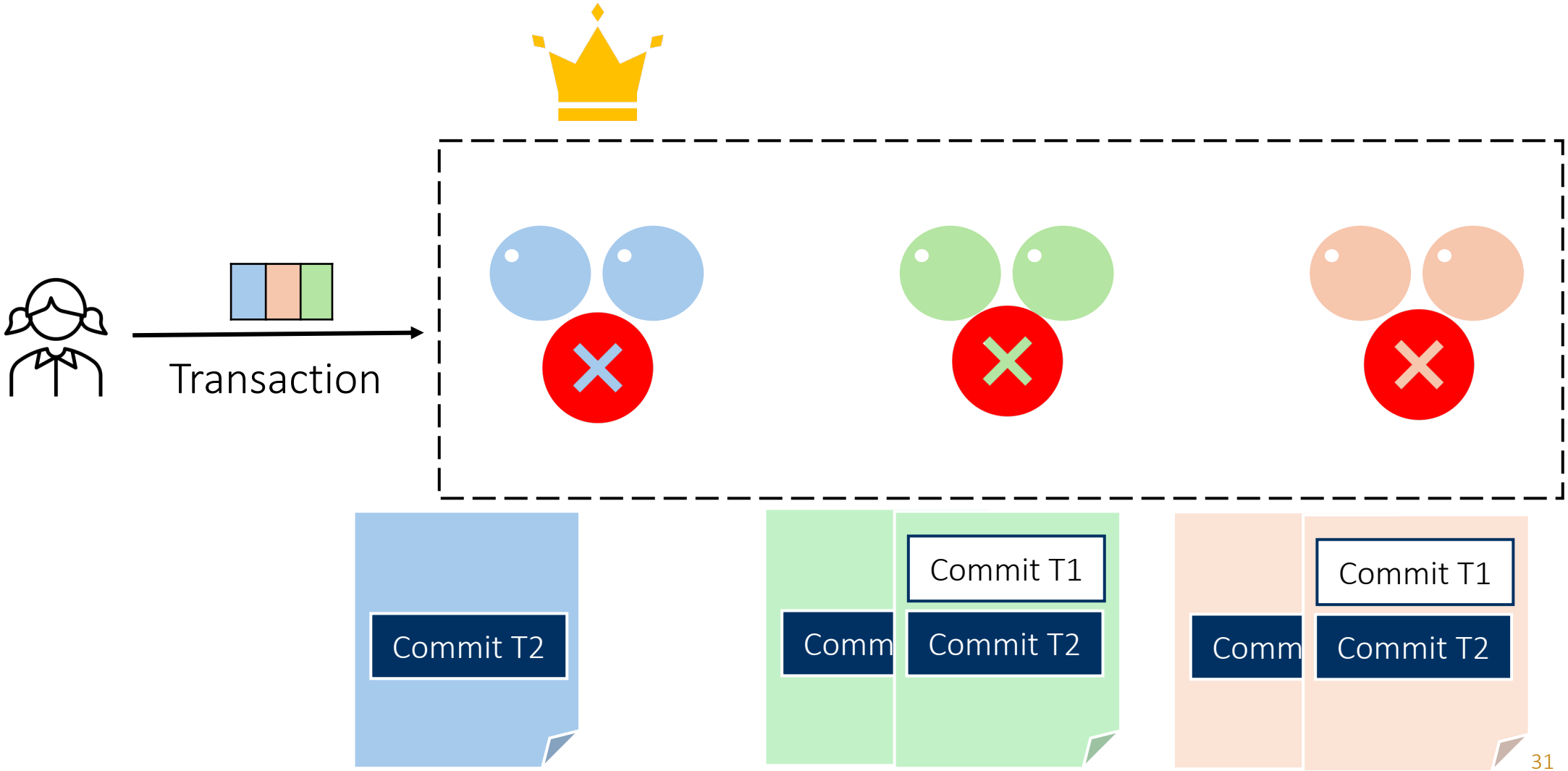
Insight: Use Cascade's Replicas for Simplifying Recovery



Insight: Use Cascade's Replicas for Simplifying Recovery



Insight: Use Cascade's Replicas for Simplifying Recovery



Cascades: High Throughput and Simple Recovery

- Provides the same consistency guarantees to the client
  - Delays notification until durability

- Cascades simultaneously achieves
  - High-throughput
  - With asynchronous logging
  - Without trading off simplicity of recovery

Cascades: Performance Preview

Builds atop Lattice*, an asynchronous logging framework from MSR
*Jose Faleiro, Jonathan Goldstein, Phil Bernstein from MSR Redmond

For highly-conflicted transactions and relative to synchronous logging on network-replicated premium-SSDs for logging, Cascades provides 160x higher throughput

Instead with high-speed ultra-SSDs (4x faster than premium-SSDs), Cascades provides 35x higher throughput

Find me at the poster session! soujanya@berkeley.edu